Technical Report: GIT-CERCS-09-06

A Characterization and Analysis of GPGPU Kernels

Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
arkerr@gatech.edu gregory.diamos@gatech.edu sudha@ece.gatech.edu

May 5, 2009

Abstract

General purpose application development for GPUs (GPGPU) has recently gained momentum as a cost-effective approach for accelerating data- and compute-intensive applications, pushed to the forefront by the introduction of C-based programming environments such as NVIDIA's CUDA, [1], OpenCL [2], and Intel's Ct [3]. While significant effort has been focused on developing and evaluating applications and software tools, comparatively little has been devoted to the analysis and characterization of applications to assist future work in compiler optimizations, application re-structuring, and GPGPU micro-architecture design.

This paper proposes a set of metrics for GPGPU workloads and uses these metrics to analyze the behavior of GPGPU programs. We report on an analysis of over 50 kernels and applications including the full NVIDIA CUDA SDK [4] covering control flow, data flow, parallelism and memory behavior. The analysis was performed using a full function emulator we developed that implements the NVIDIA virtual machine referred to as PTX (Parallel Thread eXecution architecture) - a machine model and low level virtual ISA. The emulator can execute compiled kernels from the CUDA compiler, currently supports the full PTX 1.3 specification [5], and has been validated against the full CUDA SDK. The results quantify the importance of optimizations such as those for branch re-convergence, the prevalance of sharing between threads, and the opportunities for additional parallelism.

I. Introduction

General purpose application development for GPUs (GPGPU) has recently gained momentum as a cost-effective approach for accelerating data- and compute-intensive applications. It has been pushed to the forefront by the introduction of C-based programming environments for highly data parallel architectures. CUDA for NVIDIA GPUs [1], OpenCL as an emerging standard [2], and Intel's Ct [3] are at the forefront of these environments. As perhaps the earliest widely available C-based programming environment for commodity GPUs, NVIDIA's CUDA programming model has dominated the GPGPU application landscape, having been successfully employed to exploit data parallelism from diverse domains such as signal and image processing [6], astrophysics [7], database acceleration [8], financial analysis [9]. Library developers have followed closely behind, with implementations of generic algorithms and data structures on GPUs [10]. Although targeted to NVIDIA's products, the CUDA programming abstractions are representative of the broader range of data parallel applications with single instruction stream multiple data (SIMD) stream execution models.

It is significant that this landscape of applications has been driven by the ready availability of highly parallel SIMD architectures. The NVIDIA GT200 architecture is reported to have up to 30 processors, each supporting 512-way multithreading and 32 wide SIMD units with dual issue data paths while the AMD r700 architecture is comprised of up to 60 processors, each with a 16 wide SIMD set of 5 wide VLIW data paths. Future generations of these processors can be expected to scale the number of processors as well as SIMD width. Central to the efficiency and effectiveness of these future architectures is an understanding of the impact of workload features such as control flow, data sharing patterns, memory referencing behavior, etc., and the use of this information to inform compiler optimizations, application level restructuring, and micro-architecture support.

This paper proposes a set of metrics for GPGPU workloads and uses these metrics to analyze the behavior of GPGPU programs. We report on a comprehensive analysis of over 50 kernels and applications including the full CUDA SDK [4] covering control flow, data flow, parallelism and memory behavior. We specifically study the impact of some key known optimizations such as branch re-convergence mechanisms and memory read coalescing to assess their importance while also attempting to identify important targets for future optimizations. The goal of this paper is the analysis of the behavior of data parallel application kernels and consequently the emphasis is different than micro-architecture centric studies such as [11], [12] (see Section V). The analysis was performed using a full function emulator we developed that implements the NVIDIA virtual machine referred to as PTX (Parallel Thread eXecution architecture) - a machine model and low level virtual ISA that is translated by the NVIDIA driver to the native device ISA at load time. The emulator is based on the development of a set of backend code analysis and binary translation tools for PTX, currently supports the full PTX 1.3 specification [5], and has been validated against the full CUDA SDK. The environment can execute GPGPU applications compiled with NVIDIA compilation flow and in fact can transparently replace the GPU device. While we use PTX to leverage the available compilation tool chains, we argue that PTX captures essential features of SIMD computation and is not restrictive in terms of the implications of our analysis for a microarchitecture implementation.

The following section provides some important background information on the CUDA programming environment, the data parallel architecture model, the PTX abstractions and the design of our emulator. Section III describes the workloads that are analyzed followed by a section that presents the various categories of metrics and the key results of our analysis. The paper concludes with a brief comparison with related work and a summary of anticipated and ongoing work.

II. Analysis Infrastructure

Our analysis infrastructure is comprised of a i) software emulator, ii) run-time, and iii) analysis modules. This infrastructure is part of a larger dynamic translation infrastructure called Ocelot. The simulator and analysis currently supports the parallel thread execution (PTX) virtual ISA of NVIDIA's CUDA programming model and target architectures. This section describes the CUDA environment and architecture and our analysis infrastructure.

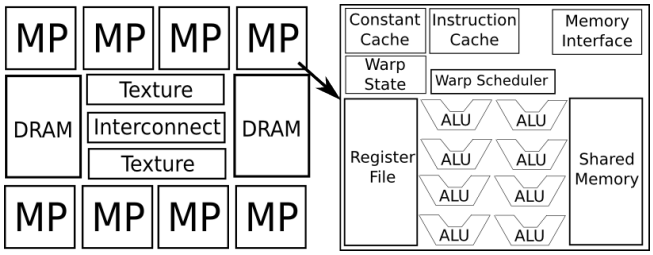

Fig. 1. Example CUDA GPU Architecture

A. The CUDA Programming Model

At a very high level, CUDA can be viewed as a set of extensions to the C/C++ programming language that allows the programmer to distinguish between highly multithreaded GPU functions (henceforth *kernels*), and native host functions. Kernels are expressed as SIMD programs and explicitly manage the GPU memory hierarchy. Kernels are managed by a series of API calls implemented by the CUDA runtime that allocate GPU memory, copy data between the host and GPU memories, launch kernels, etc. The CUDA compiler compiles kernels to PTX which is an intermediate virtual ISA that is translated to the native ISA at load time. The compiled PTX kernels (and possibly several native representations) are packed into a fat binary structure containing separate entries for each kernel and is stored as a static array along with the native C/C++ code to be executed in the host. As of CUDA 2.1, the PTX textural representation is inlined directly in unicode format in the fat binary in the source code. The resulting source file is a pure C/C++ source file, which is then passed to a native compiler such as gcc or msvcc. The final source file contains references to the CUDA API functions, and must be linked against an implementation of the CUDA runtime to be executed.

B. The PTX Machine Model

The process of invoking a GPU kernel involves the host application setting up and launching a kernel, which is stored internally in PTX and which is then translated to the native GPU binary implementation at launch. Figure 1 shows an abstraction of an NVIDIA Tesla architecture and the associated hierarchy of thread abstractions. For example, a specific instance of the architecture is comprised of several multiprocessors, each with eight-wide SIMD data paths clocked at 4x the frequency of the instruction fetch logic.

Threads and Warps. The basic unit of execution in PTX is a light-weight thread. Threads are grouped into SIMD units called *warps*. The warp size is implementation dependent, and available to individual threads via a pre-set register. The register file and shared memory structures are shared among all warps on a multiprocessor. The warp size in our analysis is 32. In order to support arbitrary control flow as a programming abstraction while retaining the high arithmetic density of SIMD operations, NVIDIA GPUs provide hardware support for dynamically splitting warps with divergent threads (distinct branch outcomes in the threads in a warp) and recombining them at explicit synchronization points. PTX allows all branch instructions to be specified as divergent or non-divergent, where non-divergent branches are guaranteed by the compiler to be evaluated equivalently by all threads in a warp. For divergent branches, targets are checked across all threads in the warp, and if any two threads in the warp evaluate the branch target differently, the warp is split and the threads contained within are serialized [13]. PTX does not support indirect branches, limiting the maximum ways a warp can be split to two in a single instruction. Fung et. al. [14] show that the PTX translator can insert synchronization points at post-dominators of the original divergent branch where warps can be recombined.

CTAs. The second level of hierarchy in PTX groups warps into concurrent thread arrays (CTAs), which are run to completion on one of the multiprocessors shown in Figure 1. The memory consistency model at this point changes from sequential consistency at the thread level to weak consistency with

synchronization points at the CTA level. Threads within a CTA are assumed to execute in parallel, with an ordering constrained by explicit barriers. CTAs also have access to an additional fixed size memory space called shared memory. PTX programs explicitly declare the desired CTA and shared memory size; this is in contrast to the warp size, which is determined at runtime.

Kernels. The final level of hierarchy in PTX groups CTAs into kernels. CTAs are not assumed to execute in parallel in a kernel (although they can), which changes the memory consistency model to weak consistency without synchronization. Synchronization at this level can be achieved by launching a kernel multiple times, but PTX intentionally provides no support for controlling the order of execution of CTAs in a kernel. Communication among CTAs in a kernel is only possible through shared read-only data structures in main memory and a set of unordered atomic update operations to main memory.

Collectively, these abstractions allow PTX programs to express an arbitrary amount of data parallelism in an application: the current implementation limits the maximum number of threads to 33,554,432 [1]. The requirement of weak consistency with no synchronization among CTAs in a kernel may seem overly strict from a programming perspective. However, from a hardware perspective, it allows multiprocessors to be designed with non-coherent caches, independent memory controllers, and wide SIMD units to hide memory latency with fine grained temporal multithreading: this structure of parallelism allows future architectures to scale via concurrent units rather than via frequency scaling.

C. Ocelot

The closed nature of GPU drivers coupled with the just-in-time compilation of PTX kernels makes analysis of the temporal properties of PTX kernels infeasible. Consequently we have developed a software emulator for PTX that can execute compiled PTX kernels. A PTX assembly file is parsed and analyzed to produce an internal representation of the program stored as a control flow graph. The parser determines the memory requirements of statically declared variables and the requested sizes of GPU specific memory structures before allocating equivalent structures in host memory. A register allocation phase is needed to convert from the infinite virtual register set used by PTX to a more manageable size to improve cacheability on the host processor.

Execution of a kernel is handled by issuing CTAs one at a time to the emulator which executes the CTA completion. The PTX execution model defines a warp as a set of threads within a CTA to execute in SIMD fashion, with explicit barrier instructions to synchronize all warps. While the number of threads per warp is an architecture-dependent constant, Ocelot is decoupled from particular implementations by setting warp size equal to the CTA size for each kernel. Divergent control flow in which some threads of a warp branch while others fall through are handled as follows. The warp is split into taken, not taken, and converge groups. The contexts describing each group, containing program counter and thread activity mask, are pushed onto a stack with inactive threads in each group predicated off. Upon reaching a compiler-inserted converge point, the stack is popped, and the next warp begins execution. After executing each path of a divergent branch, the taken and not taken paths will have been popped, and the converge context will continue execution. An example of this process is shown in Figure 2. The converge point is determined through dominator analysis of the kernel's control flow graph.

Most CUDA applications interleave native code segments with PTX kernel calls. Interoperability with these applications is provided by a runtime component of Ocelot that implements the CUDA runtime API but makes calls into the Ocelot emulation framework rather than dispatching kernel invocations to the GPU driver.

III. WORKLOADS

Using the preceding infrastructure we analyze three major categories of workloads - the CUDA SDK [4], a GPU implementation of the VSIPL API [6], and a retail Risk Inference and Analysis Application that we refer to as RIAA. All workloads were compiled with the NVCC 2.1 CUDA frontend with the GCC 4.3.2 compiler backend, and the emulated execution results were validated with outputs from the execution of the same binaries on a 280GTX NVDIA processor.

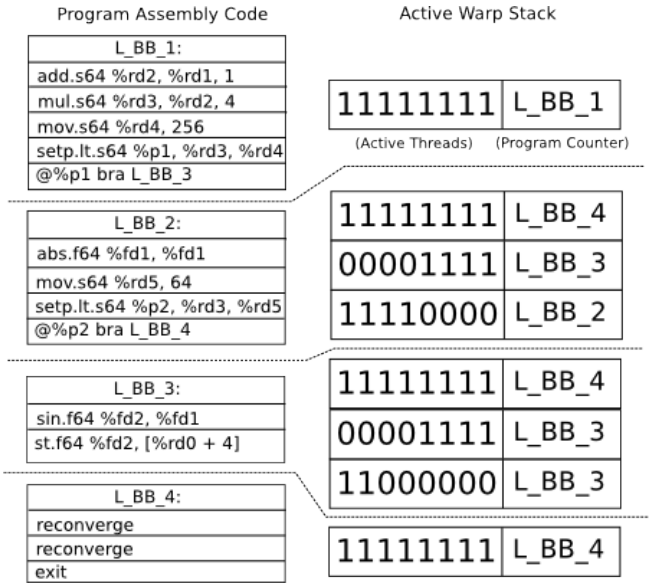

Fig. 2. Example Divergent Branch Handling

SDK Tests	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Aligned Types	12	256	64	60957	6250	4
Async API	1	512	32768	458752	0	1
Bandwidth Test	0	0	0	0	0	1
Clock Test	1	256	64	16768	1984	6
Device Query	0	0	0	0	0	1
C++ Integration	2	7.17	1	34	0	1
Atomics	1	256	64	2304	0	1
CUBLAS	1	512	81	215271	9072	9
CUFFT	14	45.55	5.42	2744	51	4
MultiGPU	1	256	32	787328	131136	3
Streams	52	512	3.29	8568	1008	3
Templates	2	47.08	1	41	0	1
Texture	2	64	4096	2031616	163840	3
Texture3D	2	256	1024	114688	6144	2
Vote	3	512	3.66	316	3	2
Template	1	32	1	21	0	1

TABLE I
SDK TEST STATISTICS

A. CUDA SDK

The CUDA SDK comprises the following groups of applications.

Regression Tests. The regression tests in the CUDA SDK evaluate the correct functionality of the unique features of CUDA. For example, there is a test for alignment of data structures in memory, asynchronous kernel launches and memory operations, GPU performance counters, atomic operations, reduction instructions, OpenGL interoperability, multithreaded host applications, multi-GPU support, and texture interpolation. These applications represent 16 out of 50 SDK applications and their basic characteristics are shown in Table I.

Building Blocks. These represent efficient implementations of low level primitives such as parallel sorting, reductions, convolution, FFTs, histograms, matrix multiplication, parallel prefix sum, data parallel scalar product, and matrix transpose. These applications represent 12 out of 50 SDK and their basic characteristics are shown in Table II.

Full Applications. The remaining 22 SDK applications represent complete applications and their basic

Building Blocks	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bitonic Sort	1	128	1	1091	176	11
ConvolutionFFT2D	15	85.31	152.6	960129	14681	3
Separable Convolution	4	141.58	304	204480	10112	5
Texture Convolution	2	192	1376	520192	2752	3
Histogram64	2	191.77	119.5	1169535	56969	4
Histogram256	5	189.45	140.8	3309550	293753	5
Matrix Multiply	1	256	40	9760	200	3
Reduction	2	127.89	32.5	41521	4354	3
Scalar Product	1	256	256	148224	16128	6
Scan	3	309.31	1	1455	102	5
Scan Large	10	249.26	8.2	28500	1782	5
Transpose	4	256	4096	802816	24576	2

TABLE II
SDK BUILDING BLOCK STATISTICS

Applications	Kernels	CTA Size	Average CTAs	Instructions	Branches	Branch Depth
Bicubic Texture	27	256	1024	222208	5120	3
Binomial Options	1	256	4	725280	68160	8
Black-Scholes Options	1	128	480	3735550	94230	4
Box Filter	3	32	16	1273808	17568	4
DCT	9	70.01	2446	1898752	25600	3
Haar wavelets	2	479.99	2.5	1912	84	5
DXT Compression	1	64	64	673676	28800	8
Eigen Values	3	256	4.33	9163154	834084	13
Fast Walsh Transform	11	389.94	36.8	32752	1216	4
Fluids	4	36.79	32.6	151654	3380	5
Image Denoising	8	64	25	4632200	149400	6
Mandelbrot	2	256	40	6136566	614210	26
Mersenne twister	2	128	32	1552704	47072	7
Monte Carlo Options	2	243.54	96	1173898	76512	8
Threaded Monte Carlo	4	243.54	96	1173898	76512	8
Nbody	1	256	4	82784	1064	5
Ocean	4	64	488.25	390786	17061	7
Particles	16	86.79	29.75	277234	26832	16
Quasirandom	2	278.11	128	3219609	391637	8
Recursive Gaussian	2	78.18	516	3436672	41088	8
Sobel Filter	12	153.68	426.66	2157884	101140	6
Volume Render	1	256	1024	2874424	139061	5

TABLE III
SDK APPLICATION STATISTICS

characteristics are shown in Table III. They cover the three prominent stock option pricing algorithms: binomial expansion, Monte Carlo, and Black- Scholes. For image processing, there are examples of bicubic interpolation, DirectX texture compression, box filters, knn denoising, and a Sobel filter. In signal processing, there are implementations of discrete cosine transform, Haar wavelet decomposition, and fast walsh transform. There are several physics simulations for particle dynamics, n-body simulation, turbulent fluid flow, and ocean wave flow. Finally, there are implementations of Mersenne Twister and Niederreiter random number generators.

B. VSIPL

The Vector Signal Image Processing Library (VSIPL) [6] is a standardized object-based signal processing API developed by the DARPA High-Performance Embedded Computing Software Initiative (HPEC-SI) [15]. VSIPL provides opaque memory abstractions to enable the development of platform-independent numerical and signal processing applications portable to systems with coprocessors and disjoint memory

Workloads	Kernels	Average CTA Size	Average CTAs	Instructions	Branches	Branch Depth
SDK	145	217.64	457.25	55884066	3504904	26
RIAA	10	64	16	322952484	23413125	16
RDG	1952	139.409	25.5738	2751900	150598	5
RDM	2237	174.558	63.0595	46448530	4082425	6

TABLE IV
WORKLOAD STATISTICS

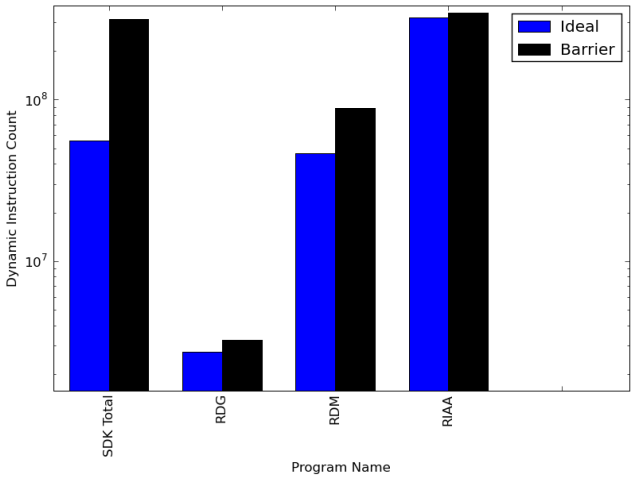

Fig. 3. Dynamic Instruction Count

spaces. GPU VSIPL [16] is an implementation of VSIPL targeting NVIDIA GPUs with numeric functions implemented as CUDA kernels. The implementation of this library is distributed with several sample applications including a range-Doppler map implementation, a common front-end processing algorithm for synthetic aperture radar (SAR) systems. This application uses GPU VSIPL's FFT functionality, itself provided by the CUDA FFT library, to process raw SAR data into an image.

C. RIAA

RIAA is a statistical forecasting application used to predict the default probabilities of loan portfolios owned by specific companies. It uses a Monte Carlo approach where a statistical distribution is sampled to produce inputs to the model subject to the constraints of an individual portfolio. The model conditionally applies a series of analyses to the initial random samples, resulting in code that is both highly data parallel and branch intensive. After the results have been generated for a given set of samples, they are aggregated together to form a statistically significant result.

IV. ANALYSIS

A. Control Flow

Activity Factor. Instructions are executed in SIMD manner across all threads of a warp. However, control flow instructions may cause threads of a warp to diverge, as some threads take a branch while others fall through. We use the term **thread activity** to refer to the average fraction of threads that are active at a given time. A kernel with no control flow or with control flow consisting entirely of unconditional branches - jumps - thread activity is 100%.

Divergent Branches. As control flow divergence results in two separate sets of threads whose execution must be serialized, the location of the synchronization point has a profound impact on thread activity and the number of dynamic instructions executed. In [14], Fung et al. show that the earliest location of thread

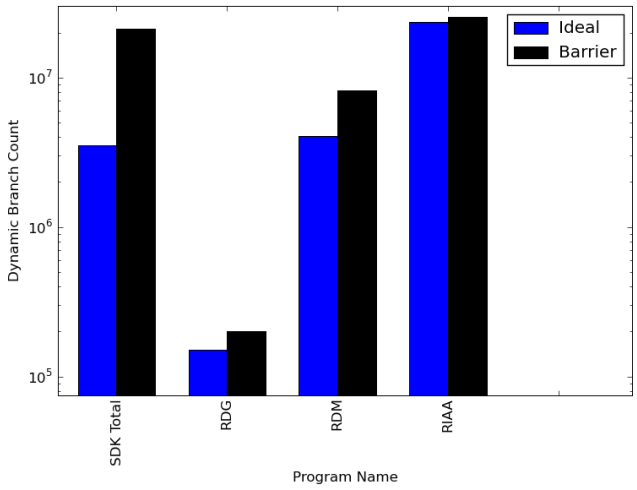

Fig. 4. Dynamic Branch Count

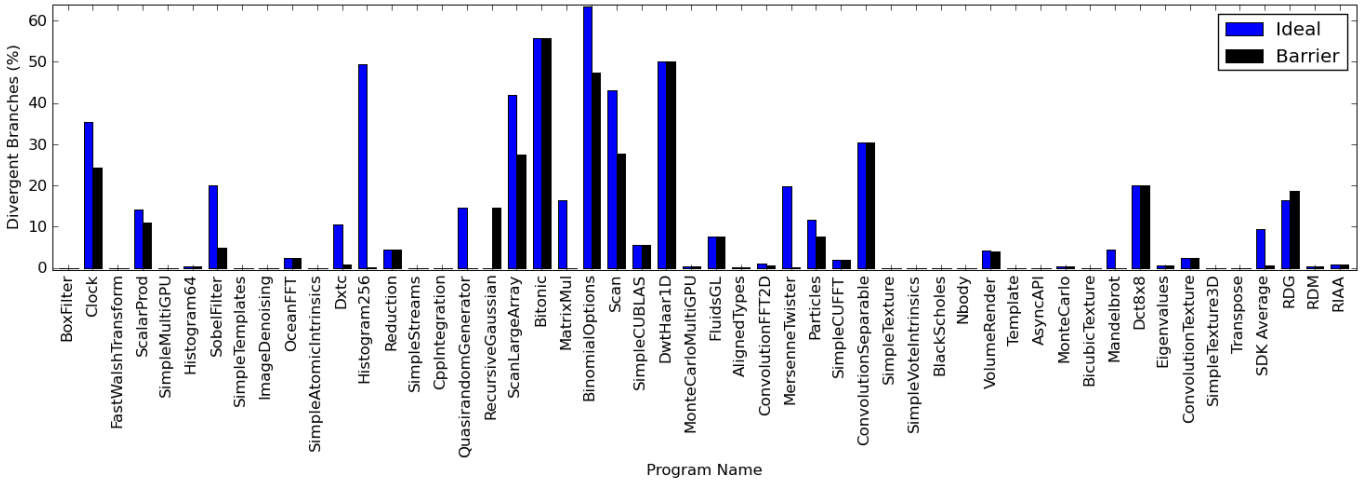

Fig. 5. Ratio of Dynamic Divergent Branches to Total Branches

reconvergence is the immediate post dominator of the branch instruction - that is, the nearest successor basic block that *must* be executed if the branch instruction is executed regardless of control path taken. Alternatively, many implementations do not reconverge threads until explicit synchronization barriers are encountered. We present thread activity and dynamic instruction counts with reconvergence at the immediate post dominator as well as at explicit synchronization barriers.

Results. Over the entire CUDA SDK, there are about 55 million dynamic instructions executed by each warp. The average warp size is 217.5 suggesting that the dynamic instruction count would be closer to 12 billion on a single threaded architecture. Of these dynamic instructions, roughly 6.5% are branches, and of those branches, 9.5% are divergent.

Moving from ideal reconvergence at the immediate post dominator to reconvergence at the next barrier instruction increases the average number of dynamic instructions significantly by a factor of 5.72 for the SDK as shown in Figure 3. The activity factor is also dramatically impacted, dropping from 85.15% to 20.35% across the SDK. The RDM workload has a similar response, increasing by a factor of 1.91. The RIAA application, on the other hand, only increases slightly by 1.06x. When looking at the SDK in detail as well in Figure 2, some applications like Histogram256 are affected dramatically by the warp convergence mechanism, while others like Bitonic are not affected at all. This behavior can potentially be explained by the fact that some applications have very few or no divergent branches and are unaffected

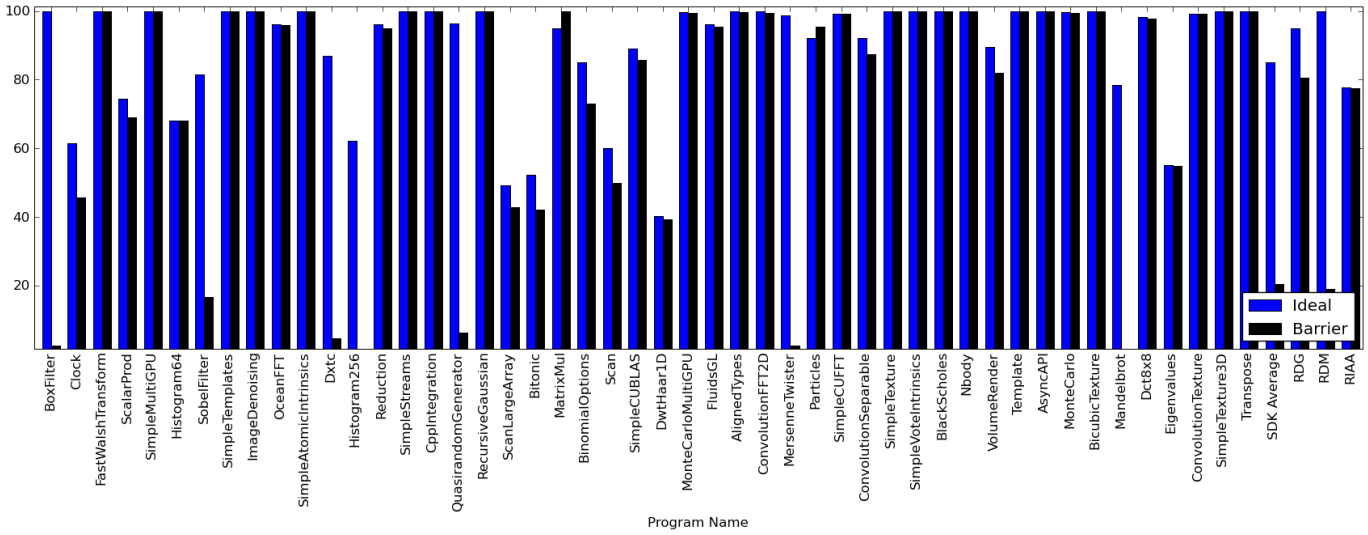

Fig. 6. Activity Factor

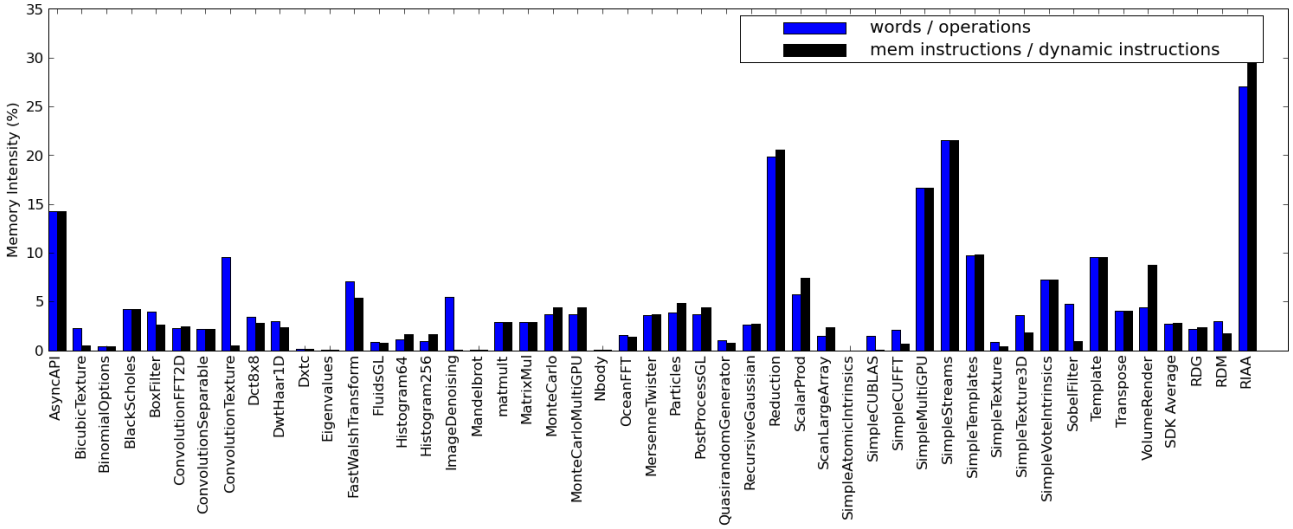

Fig. 7. Memory Intensity

by the divergence mechanism.

However, this is not true for all cases. The RIAA application has many divergent branches which dramatically affect its performance, yet moving to the barrier reconvergence scheme hardly impacts its behavior. This can be explained by examining the source code implementation, which includes barriers after most control structures. Intuitively, reconvergence at the immediate post dominator is an efficient scheme that can be implemented using compiler analysis. Barrier reconvergence, on the other hand, puts the programmer in charge: applications with barriers inserted after divergent branches converge will perform well, while applications without barriers and many divergent branches will gradually degenerate into a serial execution over time. Figure 5 shows that the ratio of divergent branches to total branches decreases by 15x using barrier reconvergence suggesting that consecutive branches are correlated, and the ideal reconvergence scheme leads to situations where warps are recombined and then immediately split again. If the programmer has intuition as to how long threads will remain divergent, it might be beneficial to allow them to specify the convergence points if the cost of splitting and recombining warps is high. However, as long as the hardware cost of splitting and recombining warps is low, it would seem

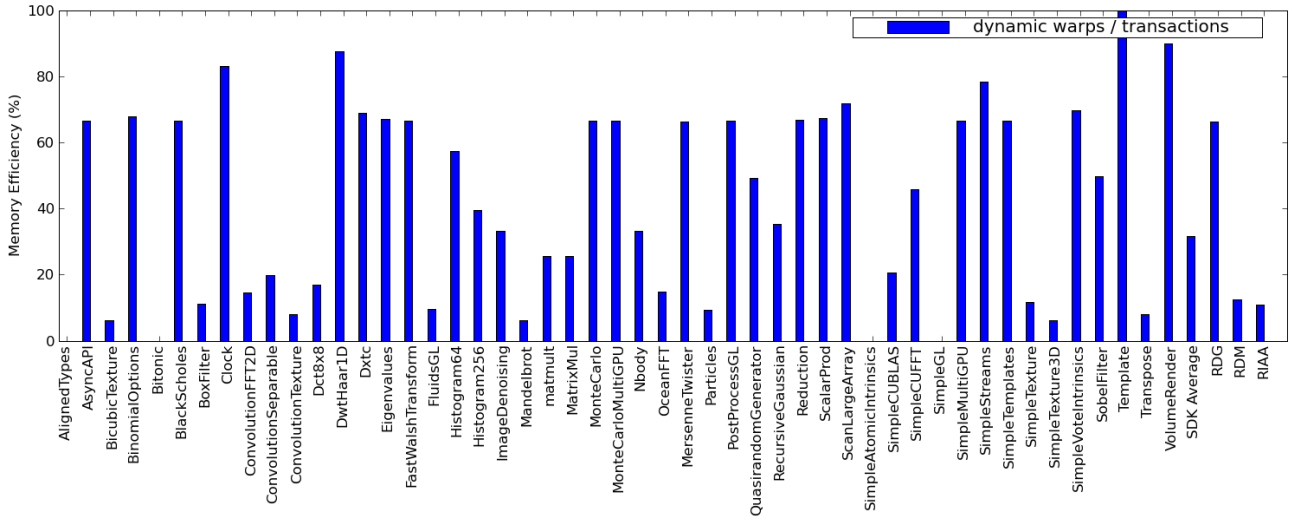

Fig. 8. Memory Efficiency

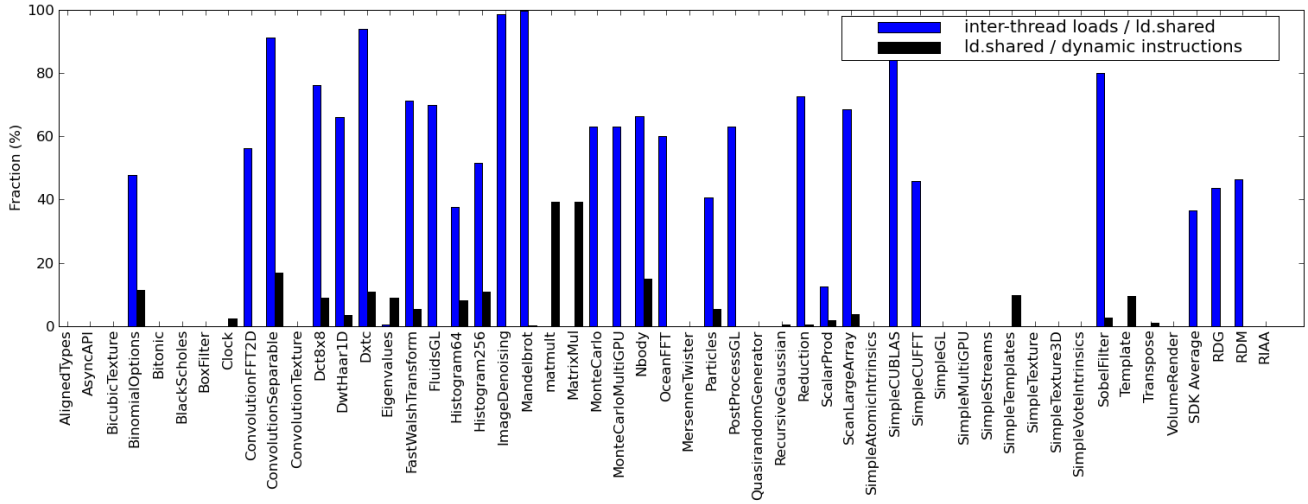

Fig. 9. Inter-thread Data Flow within shared memory.

preferable to converge as soon as possible at the immediate post dominator, since this scheme performs at least as well and often significantly better in terms of activity factor and dynamic instructions as the barrier scheme in all applications that we have examined.

The control flow behavior of Particles is dependent on the computation required for a given ball. If a ball collides with a wall of the container, then it must be handled by a procedure distinct from ball-to-ball collisions. These two special cases lead to branch divergence from the main execution loop. Based on the measured branch divergence of 11.8%, we assume collisions occur roughly $1/10$ of the time. This is a relatively high percentage of divergent branches, yet the activity factor is not dramatically impacted, remaining at 92.0%. This suggests collision handling is a significant portion of the overall computation, but it happens infrequently enough that the application is still suitable for SIMD architectures.

Recommendations. Several applications such as Particles, Mandelbrot, and RIAA exhibit control flow behavior that includes handling of special cases that necessarily result in lower thread activity. Microarchitectural support for this type of application may include algorithms for dynamic warp reformation are proposed by Fung, et al. [14] to improve utilization in which collections of threads from various warps enter a different program phase. Other applications exhibit correlated branches, and compilers and binary

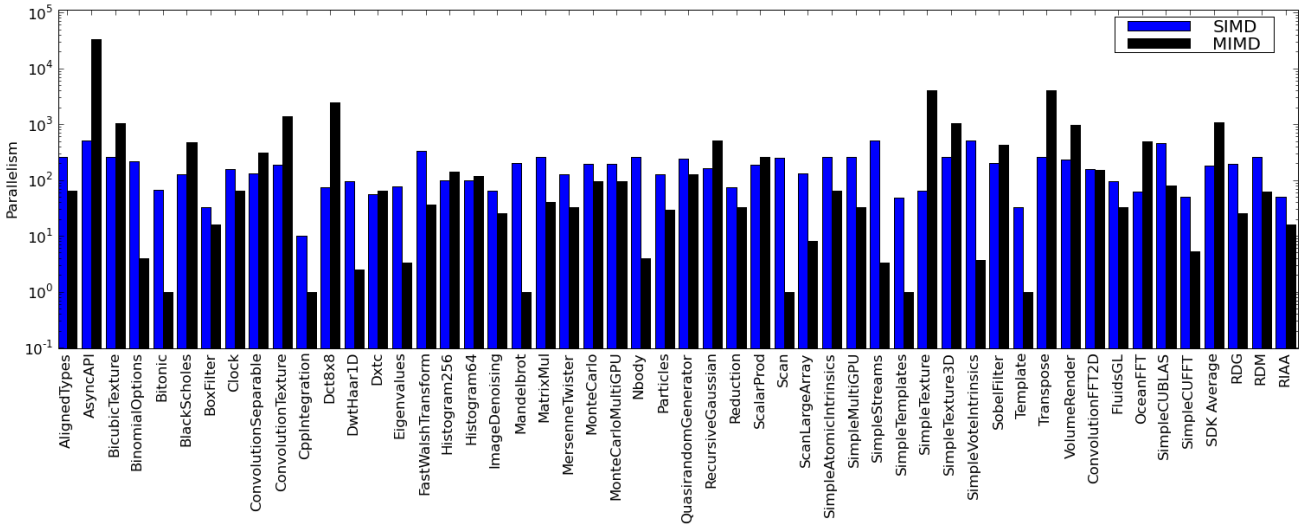

Fig. 10. SIMD and MIMD Parallelism

translators may identify opportunities in which thread reconvergence may be suggested by control flow analysis but not likely to affect thread activity. This could take the form of hot path-oriented optimizations by static compilers or dynamic translators.

B. Memory

To characterize the intensity of memory operations on PTX programs, we define the metric **memory intensity** as the number of words (in a thread) loaded or stored by accesses to global memory to the number of dynamic instructions multiplied by the average number of active threads. For the purposes of this metric, texture sampling is counted as a load with each thread fetching four 32-bit words. Memory intensity for all workloads is presented in Figure 7.

PTX defines six address spaces: constant, global, local, param, shared, and texture. Constant, param, and texture memory are read-only and backed by a cache [1]. Global memory is the largest block of memory in the memory hierarchy and also that with the largest latency. The PTX memory model enables multiple threads accessing words in the same 64- or 128-byte segment of global memory in the same load or store instruction to coalesce these accesses into a single transaction. On a GPU, this may result in one or two memory operations depending on width of the address bus and size of the transaction. Scatter operations in which each thread accesses a word in a unique segment result in one transaction per segment greatly reducing useful memory bandwidth on real-world platforms.

To characterize spatial locality of operations to global memory, the CUDA memory coalescing protocol for compute capability 1.2 was implemented [1]. To decouple our analysis from particular implementations, warp size was chosen to be equal to the number of threads within the CTA, and segment size was selected to be equal to $warpsize \times 4$ bytes. The addresses of each thread's memory references were mapped to segments, and the number of segments required to satisfy each access was accumulated. The number of warps attempting memory accesses was also accumulated. Memory bandwidth utilization approaches theoretical peaks as **memory efficiency**, the ratio of warps to transactions, approaches 100%. As the number of transactions required to satisfy a warp's loads or stores increase due to scattered access patterns, this ratio decreases. Memory efficiency for the CUDA SDK, RDM, and RIAA applications are presented in Figure 8.

Results. Applications in this workload exhibit low memory intensity. The CUDA SDK average memory intensity is 2.70%, and the RDM application's is 3.02%. This indicates the possibility they are compute bound. Applications with particularly high memory intensity include several regression tests from the CUDA SDK that intentionally stress the memory system as well as the outlying RIAA application in

which memory intensity is 27.1%. The CUDA SDK average memory efficiency of 31.0% indicates, on average, three memory transactions are required to satisfy each warp's memory accesses. Among these workloads, applications with the least memory efficiency tend to be those that rely heavily on texture sampling.

The RIAA benchmark is very intensive in terms of accesses to global memory, with accesses to memory constituting 26% of all operations making it almost certainly memory bound. Additionally, memory efficiency is quite low indicating many scattered memory references per warp. It is unclear from this analysis what cache size would be required to achieve high hit rates.

Recommendations. Several applications exhibit low memory efficiency. Of these, texture caches on architectures implementing the CUDA programming model alleviate certain incoherent access patterns, yet these read-only data structures do not affect applications with intensive scattered writes. The relaxed memory coalescing rules of CUDA compute capability 1.2 greatly reduces the impact of misaligned loads and stores. Analysis with Ocelot suggests limits in scalability of memory controllers, as CUDA SDK applications require three memory transactions per access on average.

C. Data Flow

Data Sharing. CTAs as described in II-B include a scratchpad memory block known as **shared memory**. Shared memory provides a mechanism through which data may be exchanged among the threads of a CTA assuming the state of shared memory is made consistent through barrier synchronizations. Workloads may wish to share data to accommodate the PTX memory model in which global memory loads or stores executed by all threads of a warp are coalesced into a single transaction. In this case, shared memory is effectively used by the CTA as a caching mechanism in which data loaded from global memory by one thread is broadcast to other threads.

Alternatively, threads within a CTA may exhibit a producer-consumer relationship in which computations of one thread are needed by another. Examples of this include reduction operators or the butterfly structures of a Fast Fourier Transform. To characterize workloads with producer-consumer CTAs, stores to shared memory are tracked to corresponding loads from other CTAs. The metric we report, **inter-thread data flow**, is the number of inter-thread loads from shared memory relative to the total number of words loaded from shared memory. This is presented for all workloads in Figure 9. To illustrate the impact of shared loads on the entire kernel, the fraction of shared loads to total dynamic instructions also appears in the figure.

Results. Several applications such as the Monte Carlo simulations (Black-Scholes and RIAA) do not make use of shared memory at all and exhibit no interdependencies among the threads. This suggests that threads may be arbitrarily arranged within CTAS possibly improving the activity factor. Other applications such as matrix multiply use shared memory to broadcast data streamed in from global memory and could be accommodated by architectures with data caches. Many applications, however, exhibit a high fraction of inter-thread loads. The CUDA SDK averages 36.7% of shared load instructions reading words produced by other threads. The RDG and RDM applications are dominated by FFTs with over 45% of shared loads constituting inter-thread data dependencies. In the Particles application, the sharing of data among threads is relatively high at 46%. This could suggest that threads whose balls collide must exchange data in order to determine the result. In any case, it introduces a synchronization requirement to the application.

Recommendations. Many applications in this selection of workloads exchange data between threads requiring synchronization and a conduit for transferring data. High-performance microarchitectures with this type of memory structure depend on a shared memory system with many banks so that all threads within a half-warp may write concurrently without stalls. Analysis tools like Ocelot enable characterizing access patterns within shared memory to detect the incidence of bank conflicts for particular banking schemes and warp sizes. This approach may suggest alternatives to shared memory such as on-chip networks to provide a better and more scalable approach to inter-thread data flow for certain classes of applications.

D. Parallelism

Unlike most machine models, PTX programs explicitly declare the number of threads and CTAs in the program, statically exposing parallelism in an application. This parallelism allows programs to be scaled to future GPU architectures simply by adding additional cores and SIMD units. For a given set of applications, it is useful to discover the limits of this scalability in order to determine how many cores can be added to a GPU before they will no longer be utilized. Towards this end, we define two metrics that capture parallelism in an application, MIMD parallelism and SIMD parallelism.

MIMD parallelism is computed as the speed up of a GPU with an infinite number of multiprocessors over a GPU with a single multiprocessor, ignoring memory bandwidth and latency constraints. It is computed by assuming that each instruction takes a single cycle to complete and dividing the total number of dynamic instructions by the dynamic instructions of the longest running CTA in a kernel. **SIMD parallelism**, on the other hand, is computed as the average activity factor of a CTA, multiplied by the number of threads in the CTA, and weighted by the number of dynamic instructions in the CTA. It is averaged over all CTAs in a kernel.

Results. The average MIMD parallelism across the SDK is 1076.46, while the average SIMD parallelism is 180.92. This is particularly interesting because it represents a reduced data size for the SDK where many applications are weakly scalable. The RDM and RIAA applications have slightly less MIMD parallelism, 62.6 and 15.83 respectively. These applications are also weakly scalable and the lower MIMD parallelism may be an artifact of their higher computational density than many of the SDK examples, several of which simply apply several instruction transformations to each data point. The SIMD parallelism of the SDK and RDM workloads are very similar, 180.92 and 258.32 respectively, yet strikingly larger than the 49.72 for the RIAA workload. The RIAA workload is limited by a combination of low activity factor and low number of threads per CTA driven by high register usage per thread; the large working set of application can be seen in the abnormally high memory intensity of the application in Figure 7. Of the 16 CTAs that we did run for this application, the execution time of each CTA was relatively constant resulting in a MIMD parallelism metric of 15.836. Together these results suggest that this application would be more suited to a multicore architecture with a large data cache per core, rather than a SIMD architecture with no caches.

Recommendations. Analysis of the Mandelbrot application suggest that each thread computes several pixels using a loop within the body of the kernel. Examining the source code confirms this observation. In general, adding a data parallel loop to the body of a kernel limits the amount of MIMD style parallelism in the application since fewer CTAs will be executed, but it also allows redundant code at the beginning of the kernel to be amortized across multiple iterations of a loop. In this case, there is no redundant code at the beginning and it probably would have made more sense for the programmer to eliminate the loop and launch extra CTAs, which would have increased the amount of parallelism in the application by a factor of 46.875x. This is an example of a programmer's decision that impacts the potential scalability of an application.

V. Related Work

GPU Simulation. The closed nature of native GPU instruction sets has resulted in a dearth of architecture simulators in the academic community, compared to simulators for general purpose processors, which are plentiful. In 2007, Fung et al. developed a timing accurate simulator (GPGPU-Sim) for an NVIDIA-like GPU built around the SimpleScalar back end [14]. Input applications were produced by modifying SPEC 2006, SPLASH, and CUDA applications to use a new API for launching compute kernels, where kernels written in C were compiled into native instructions and launched using a SPMD model with multiple threads similar to CUDA. The core of SimpleScalar was modified to include different modes for simulating native code and GPU kernels and to switch between modes on kernel invocations. Fung et al. used this simulator to evaluate several different schemes for warp formation in NVIDIA-like GPUs. Recently, this simulator has been extended to support PTX applications as well [12].

Barra. Recently, Collange et al. have developed Barra, a functional simulator for the native NVIDIA G8x and G9x instruction sets [11]. With a reimplementation of the low level CUDA runtime API, they produced a functional simulator capable of running CUDA applications. To the best of our knowledge, so far, Barra has been successfully tested with 15 of the 50 CUDA SDK applications.

Compared to Ocelot, Barra simulates the native GPU instruction set, while Ocelot executes the PTX virtual machine directly. Similarly GPGPU-Sim focuses on the architecture evaluation of NVIDIA-like GPUs, while Ocelot is more concerned with program and compiler analysis; we believe that this distinction makes Barra and GPGPU-Sim more suitable for the evaluation of specific architectural modifications to existing GPUs via the addition of detailed timing models, while Ocelot provides more insight into the high level characteristics of a CUDA application and offers avenues for evaluating applications on architectures other than NVIDIA-specific GPUs.

VI. Conclusions

GPGPUs have emerged as (relatively) low-cost commodity compute accelerators. The recent surge of interest has resulted in the exploration of the acceleration of compute-intensive data parallel components of a wide variety and range of applications. However, there has been relatively little reported work on GPGPU workloads and the analysis of workloads. This is due in part to the emerging and consequently proprietary nature of the GPGPU instruction sets and the lack of public domain open source tools for compilation, analysis and emulation. This paper reports on an infrastructure for the analysis of data parallel GPGPU applications. The infrastructure currently supports NVIDIA's PTX virtual ISA. It is validated against the full CUDA SDK, a DoD standard signal and image processing library and a computationally intensive risk assessment application. We propose several metrics for characterizing these kernels and report on the analysis of the proposed workloads.

Acknowledgement

The authors gratefully acknowledge the generous support of this work by LogicBlox Inc. and NVIDIA Corp. both through research grants, fellowships, as well as technical interactions, and equipment grants from Intel Corp. and NVIDIA Corp.

References

[1] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*, 2nd ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[2] K. O. W. Group, *The OpenCL Specification*, December 2008. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[3] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou, "Ct: A flexible parallel programming model for tera-scale architectures," *Intel Technology Journal*, vol. 11, no. 4, October 2007.

[4] NVIDIA, *NVIDIA CUDA SDK 2.1*, 2nd ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[5] ——, *NVIDIA Compute PTX: Parallel Thread Execution*, 1st ed., NVIDIA Corporation, Santa Clara, California, October 2008.

[6] D. Schwartz, R. Judd, W. Harrod, and D. Manley, "Vsipl 1.3 api," VSIPL Forum, Tech. Rep., 2008.

[7] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande, "N-body simulation on gpus," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 188.

[8] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 511–524.

[9] I. Buck, "Gpu computing with nvidia cuda," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. New York, NY, USA: ACM, 2007, p. 6.

[10] M. Harris, S. Sengupta, and J. Owens, *Parallel Prefix Sum (Scan) in CUDA*. Addison Wesley, 2007.

[11] S. Collange, D. Defour, and D. Parello, "Barra, a modular functional gpu simulator for gpgpu," Tech. Rep. hal-00359342, 2009.

[12] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Boston, MA, USA, April 2009.

[13] M. Simon and L. J. Erik, "Method and system for programmable pipelined graphics processing with branching instructions," Tech. Rep. 6947047, September 2005. [Online]. Available: http://www.freepatentsonline.com/6947047.html

[14] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420.

[15] D. Campbell, "The high performance embedded computing software initiative: C++ and parallelism extensions to the vector, signal, and image processing library standard," 2005, pp. 278–283.
[16] ——, "Vsipl++ acceleration using commodity graphics processors," in *HPCMP-UGC '06: Proceedings of the HPCMP Users Group Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 315–320.